

DBTLAB – Exercise 2: Adaptive Replacement Cache

Viktor Rosenfeld

based on material from Volker Markl, Alexander Alexandrov, Jonas Traub, Martin Kiefer

Course registration

- Register by Monday morning (November 6, 9:00) in MOSES
- **We cannot help you if you miss this deadline!**

Course calendar

DBTLAB [WiSe 2023/24]

Heute ◀ ▶ November 2023

Drucken Woche Monat Terminübersicht

Mo	Di	Mi	Do	Fr	Sa	So
30	31	1. Nov.	2	3	4	5
1: Table Page Layout			(12:00PM) 2: Adaptive Replacement Cache			
6	7	8	9	10	11	12
1: Table Page Layout			2: Adaptive Replacement Cache			
13	14	15	16	17	18	19
2: Adaptive Replacement Cache			(12:00PM) 3: Buffer Pool Manager			
20	21	22	23	24	25	26
2: Adaptive Replacement Cac			3: Buffer Pool Manager			
27	28	29	30	1. Dez.	2	3
3: Buffer Pool Manager			(12:00PM) 4: Operators I (Table and Index Scan)			

Terminanzeige in der Zeitzone: Mitteleuropäische Zeit - Berlin

Google Kalender

- Contains lectures, code releases, & deadlines.

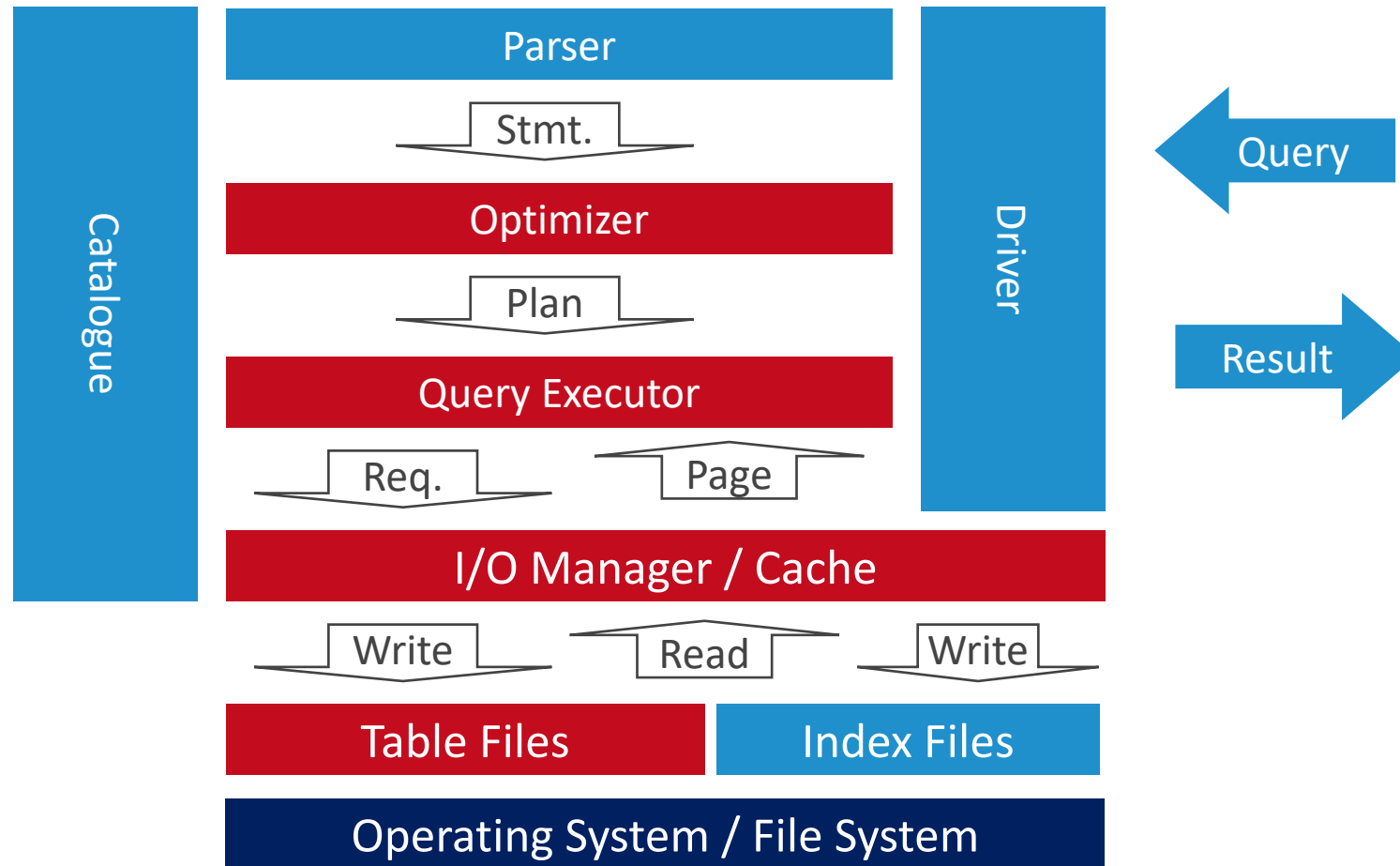
Follow-up from last lecture

- Data (records, attributes, variable) data is never deleted)
- Instead, only the tombstone bit is set (slide 23 of last week)
- Iterators have to skip these tuples
- Update = deletion and insertion

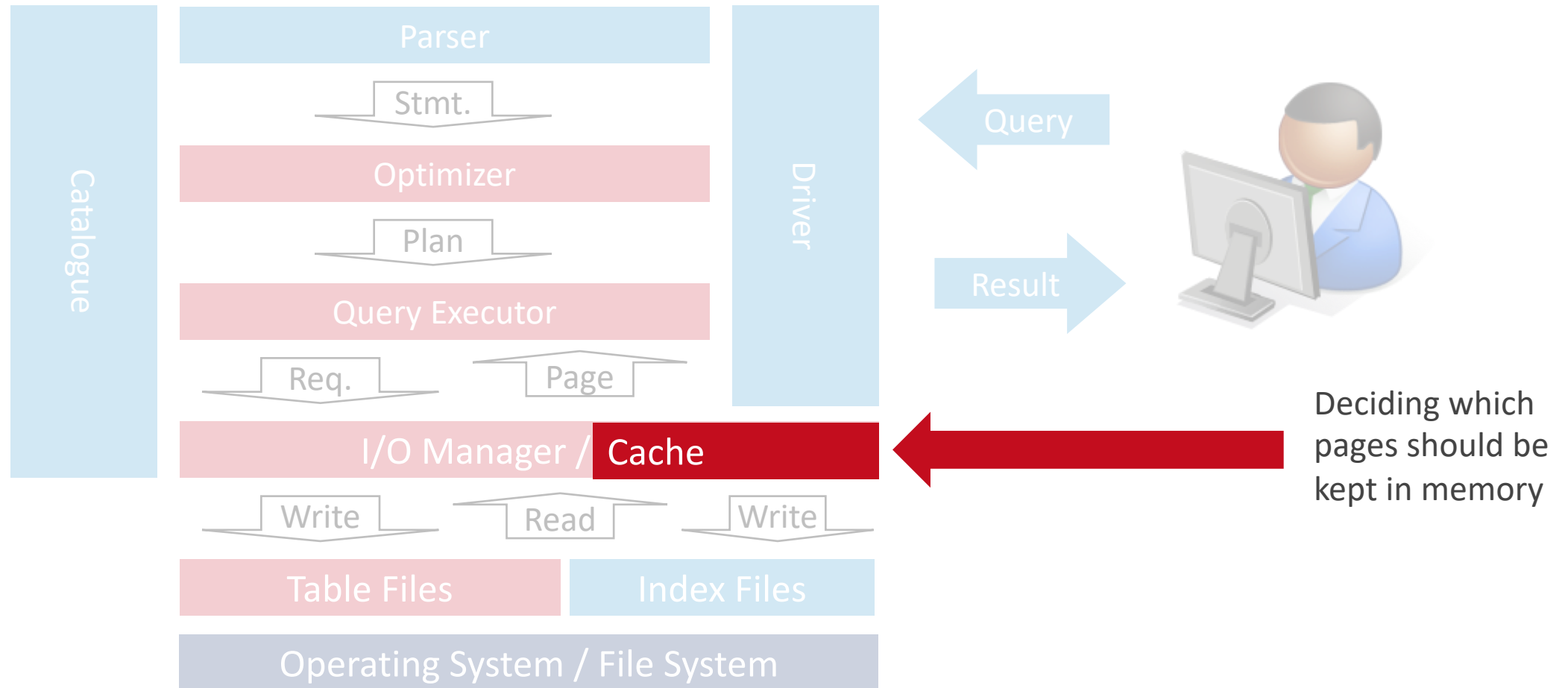
Follow-up from last lecture

- CHAR is fixed length
 - Stored directly in the record
 - Size of record includes number of bytes for the CHAR field
- VARCHAR is variable length
 - Stored in the variable length chunk area
 - Size of record includes 8 bytes for pointer

Basic system architecture

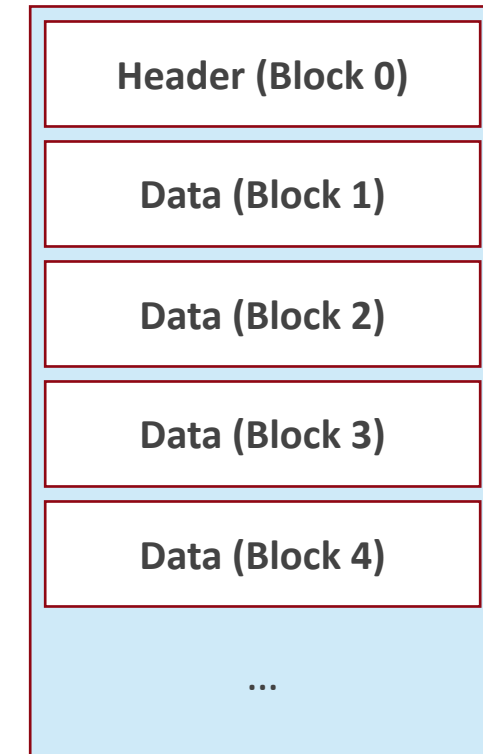


Where are we today?



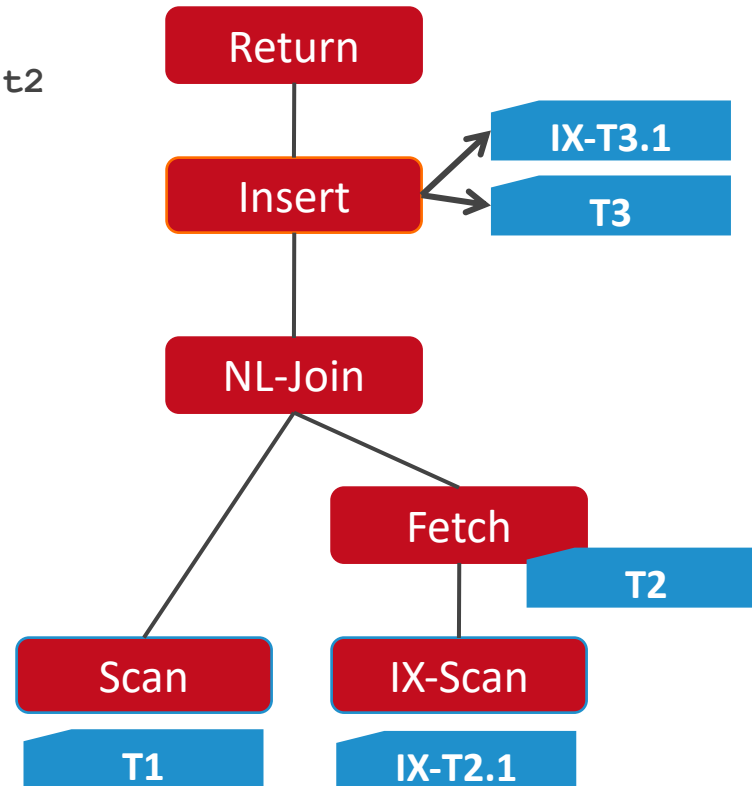
Remember: Tables and Pages

- Each table is a collection of pages (or blocks).
- Every time data from the resource is accessed, the page containing the data is accessed.



When is data in pages accessed?

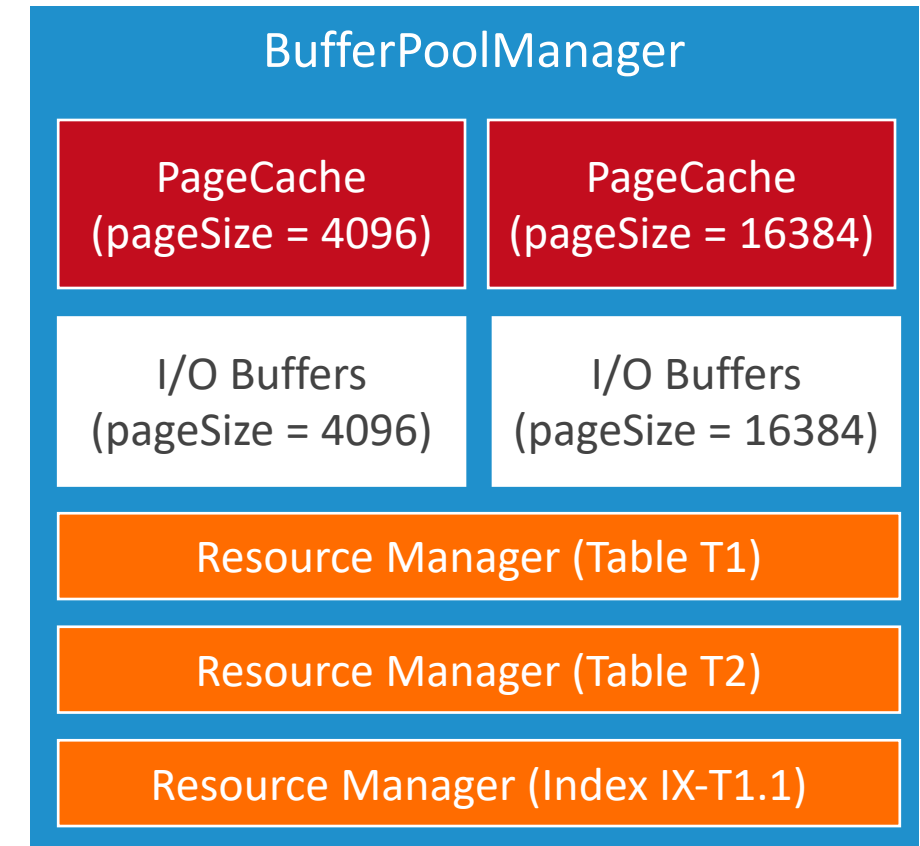
```
INSERT INTO T3
SELECT t1.a, t2.b
FROM tab1 t1, tab2 t2
WHERE t1.p = t2.f
AND t1.num > 42
```



- Scan:
 - Go over all table pages in T1
- IX-Scan:
 - Get index page #n from IX-T2.1
- Fetch:
 - Get tuple from a table with RID from T2
- Insert:
 - Insert into table T3, generate RID
 - Add key/RID to index IX-T3.1
- Also: Delete / Update

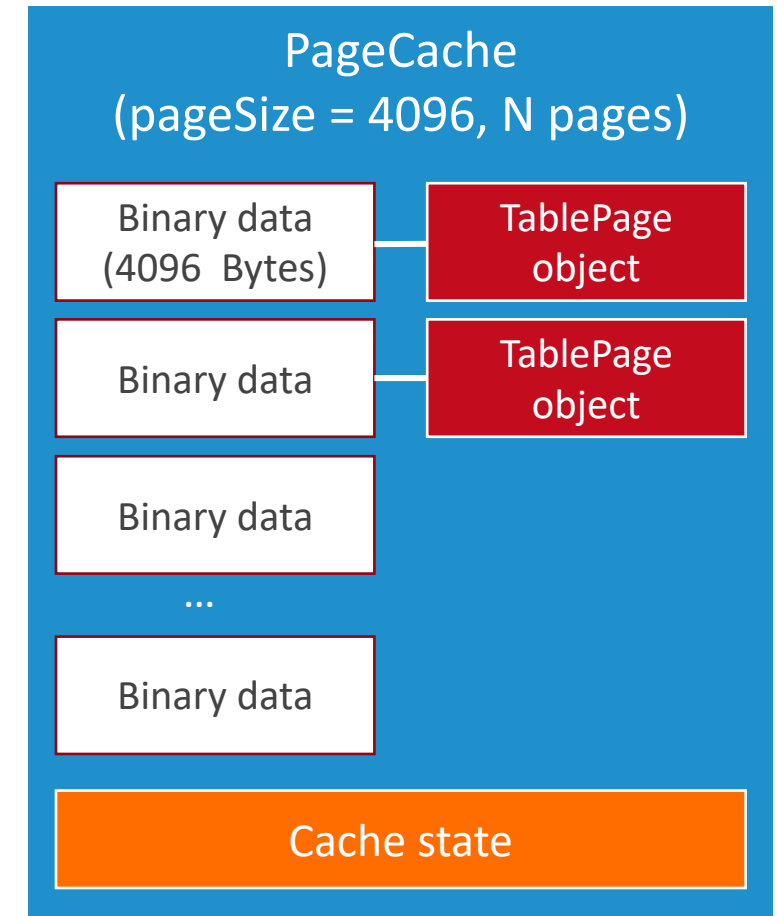
Buffer Pool Manager

- All pages are accessed through the *Buffer Pool Manager*.
- Manages multiple caches (one for each page size).
- Contains I/O buffers for reading / writing pages.
- Manages multiple resource managers (one for each table and index).
- Knows which resource (specific table or index) has its pages in which cache (depending on the page size).
- Checks the cache when a page is requested.
- Loads pages that were not in the cache.
- Resources are identified by a numeric resource ID.
 - Do not confuse with RID (row ID).
- The page size depends on the resource.



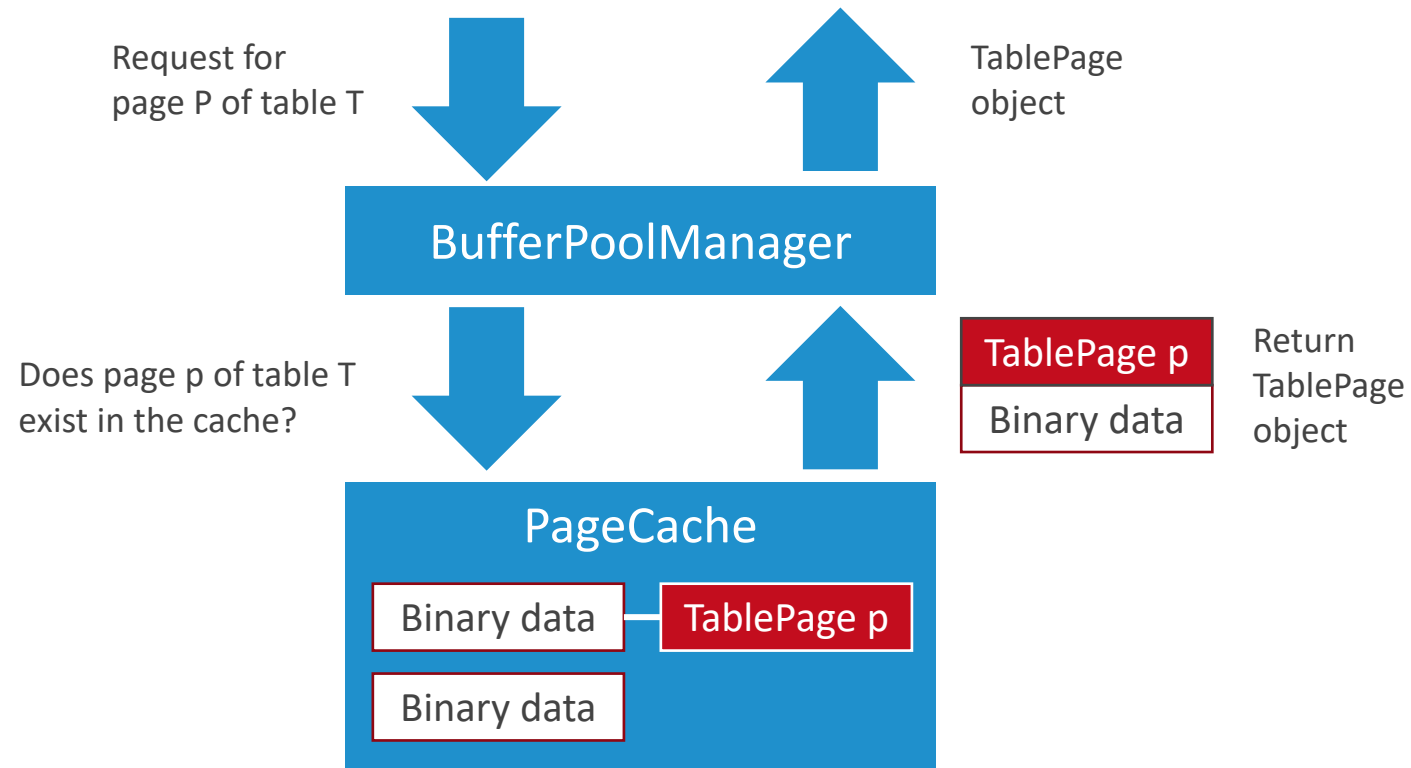
PageCache

- The PageCache owns the byte arrays which make up the cache.
 - Byte arrays are allocated when the PageCache object is created.
 - The PageCache object always uses a fixed amount of memory.
 - Once created, OutOfMemoryExceptions should not occur.
 - Size and number determined by the factory method.
- When a page is added to the cache, the wrapped byte array replaces a byte array in the cache.
 - The byte array is now associated with a TablePage object.
- Additional information to manage eviction.



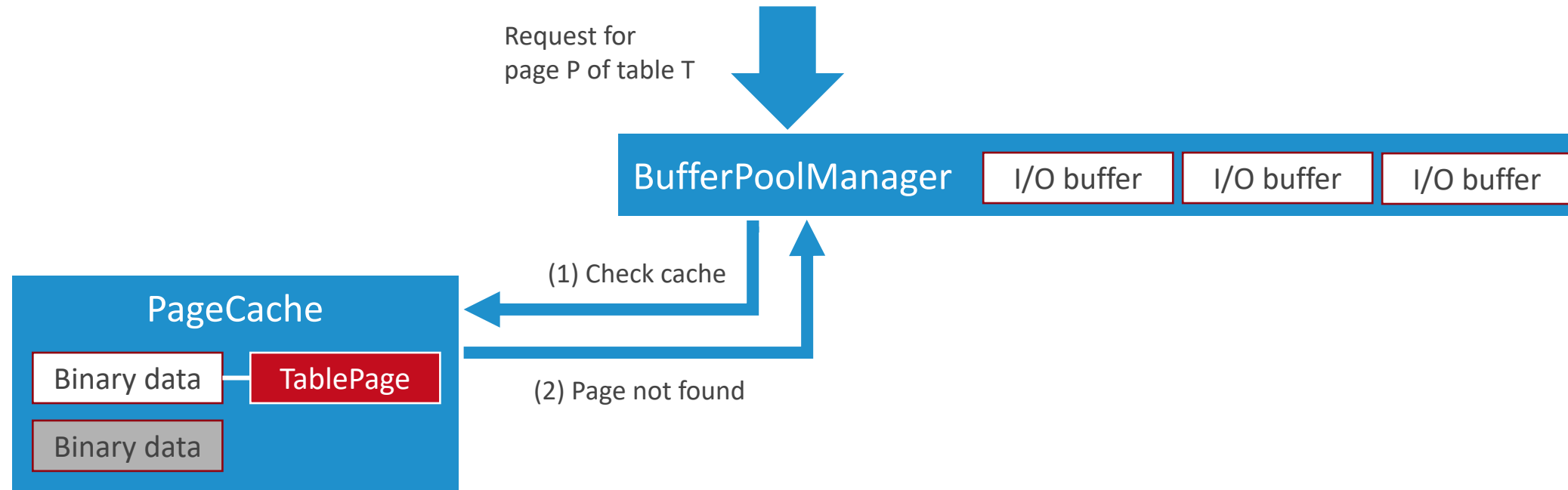
Retrieval of pages from the cache

- Case 1: Page exists in the cache (cache hit).



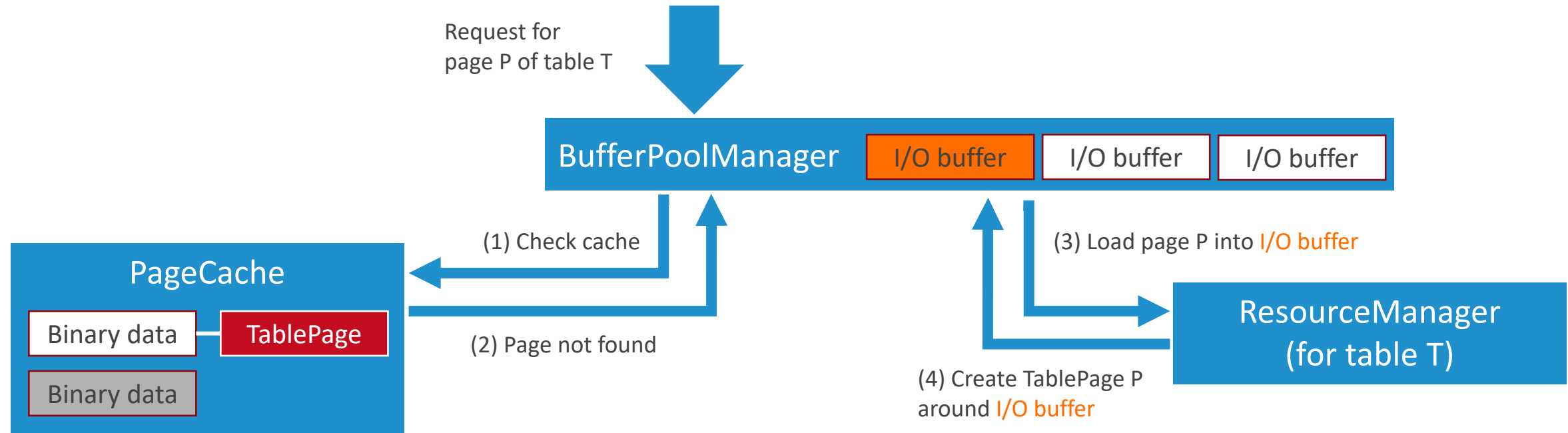
Retrieval of pages from the cache

- Case 2: Page does not exist in the cache (cache miss).



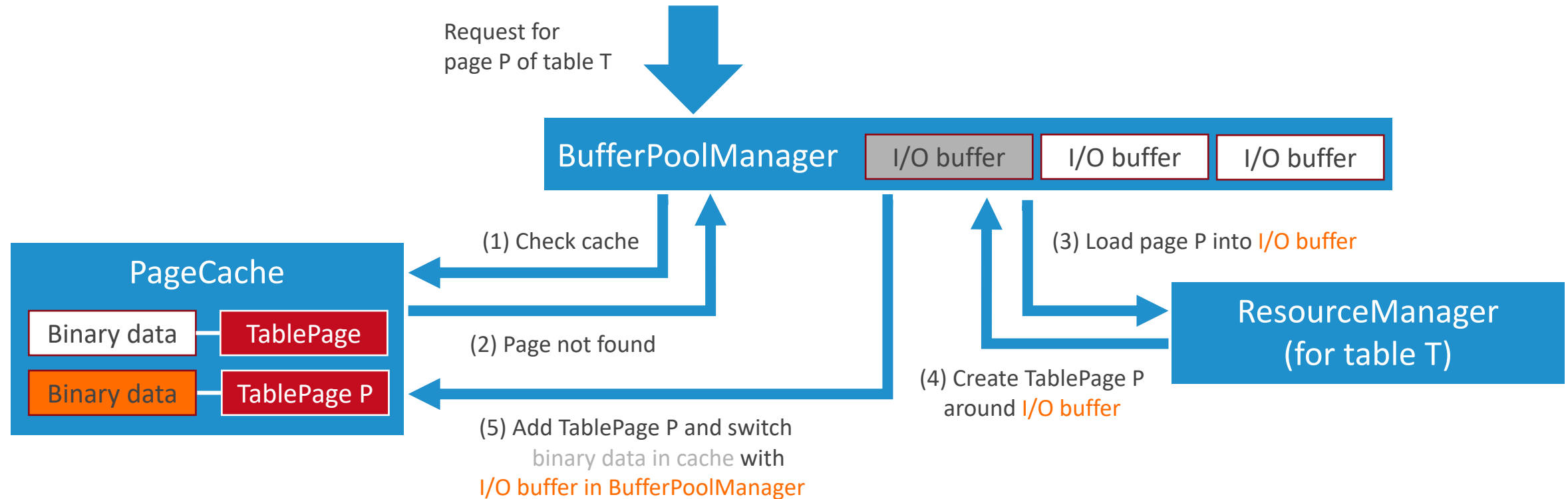
Retrieval of pages from the cache

- Case 2: Page does not exist in the cache (cache miss).



Retrieval of pages from the cache

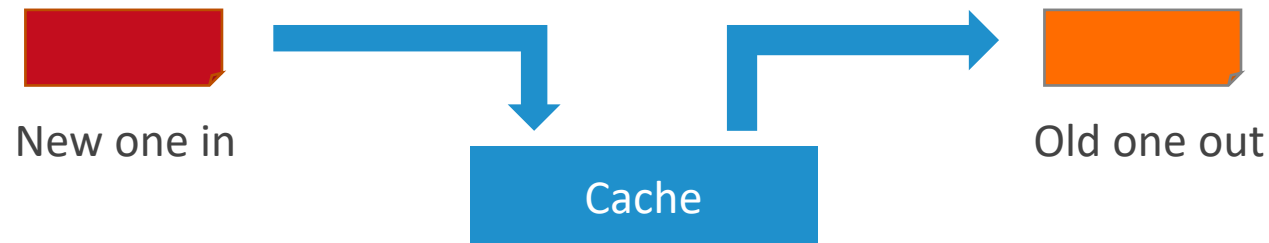
- Case 2: Page does not exist in the cache (cache miss).



Cache eviction

Cache eviction

- When a new page is inserted, an old page must be evicted.



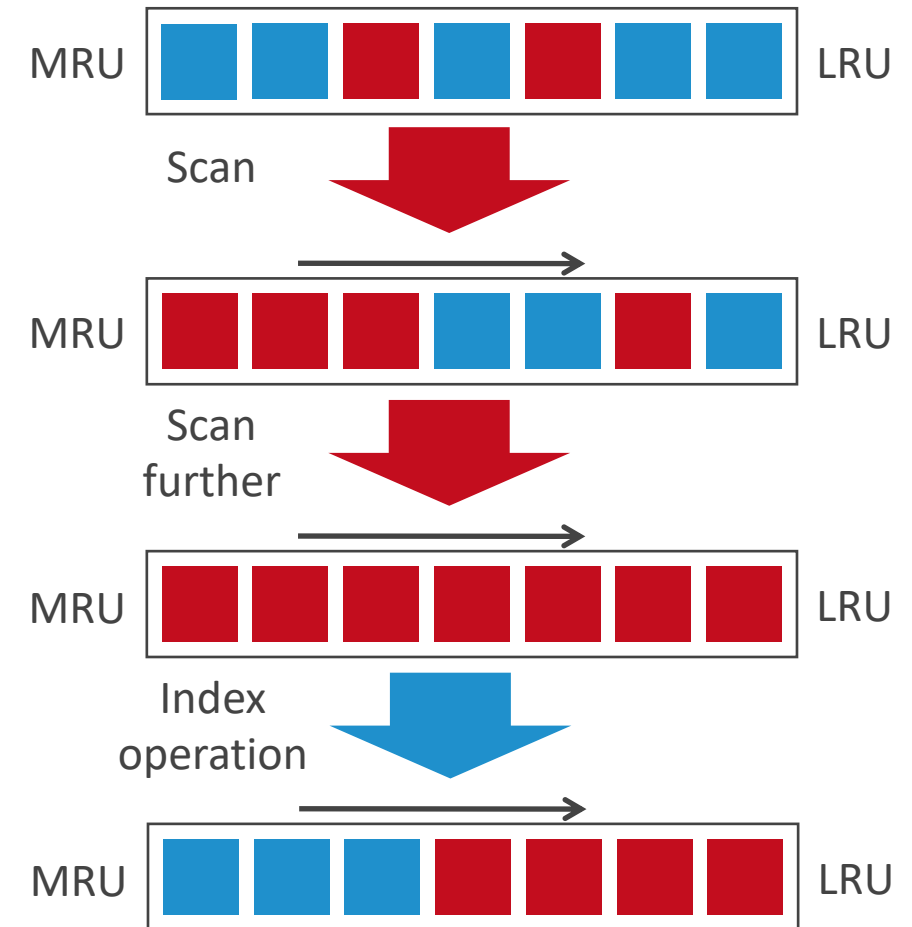
- Typical Strategies
 - Random → Surprisingly OK.
 - First-in-first-out (FIFO), Last-in-first-out (LIFO) → Bad for DBMS.
 - Least recently used (LRU) → Very common; emphasizes data that was used a lot recently.
 - Least frequently used (LFU) → Degrades when data was accessed a lot in the past.
 - CLOCK → Simple LRU approximation.
- We will use: **Adaptive Replacement Cache (ARC)** strategy
 - Variation of LRU

Scan resistance

- LRU cache and mix and of Scan and Index operations
 - Index pages are accessed frequently.
 - We want to keep them in the cache.
- Table scan replaces all pages in the cache.
- Index lookup has to load index pages in the cache again.
- Index pages where loaded just after they were evicted.



MRU = Most recently used (page in the cache)



Different caches for scan resistance?

- Scan resistance can be solved by using different caches for sequential scan operations and random lookup operations.
 - Prefetch cache for scans.
 - LRU cache for indexes.
- Has however some problems:
 - What size of the total memory should be dedicated to which cache?
 - Both caches must be checked for pages, or a resource must be strictly in one cache.
 - Tuning and runtime overhead...
- Better combine recency and frequency in a single cache.

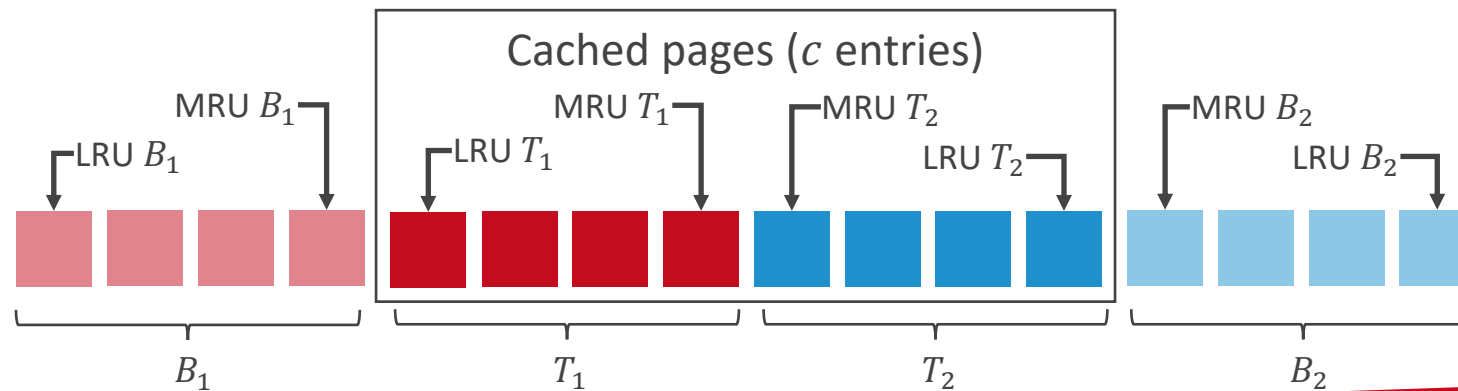
Pinned pages

- A page can be pinned in the cache.
- Pinned pages behave like normal pages, but when it becomes a candidate to be evicted, they remain inside the cache.
 - Evicted is always the LRU non-pinned page.
- Some operators access the same page repeatedly and the page should remain in cache:
 - Index-Scan accesses repeatedly the root element of a B+ tree.
- Some Operators need to make sure that the page remains valid for a while.
 - Table Scan should make sure that the page is not evicted while an iterator over the tuples is used.

Adaptive Replacement Cache

Top and bottom lists

- Split cache of size c into two *top lists*:
 - T_1 : *Recent* entries, which have been accessed only once in the past.
 - T_2 : *Frequent* entries, which have been accessed at least twice in the past.
 - Lists track resource ID and page number to identify a page.
 - Entries correspond to a byte array (and associated page) in the cache.
- Additionally track evicted pages in two *bottom lists*.
 - B_1 : Entries which have been evicted from T_1 but are still tracked.
 - B_2 : Entries which have been evicted from T_2 but are still tracked.
 - Track resource ID and page number but no associated byte array.

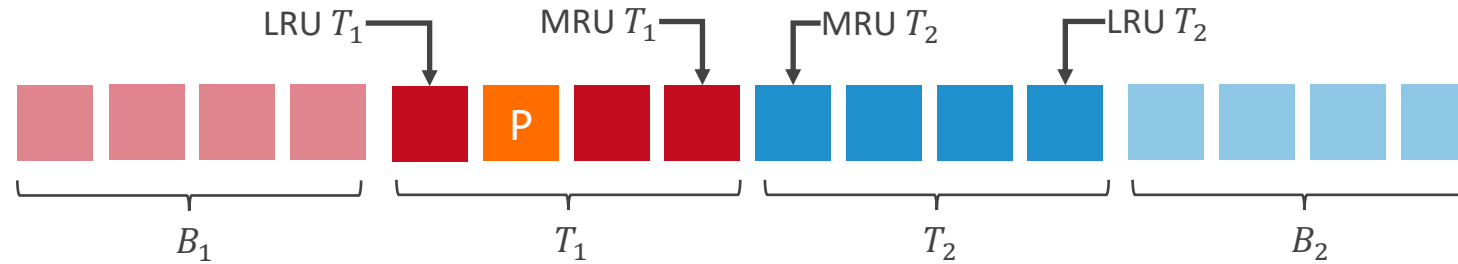


Adaptation

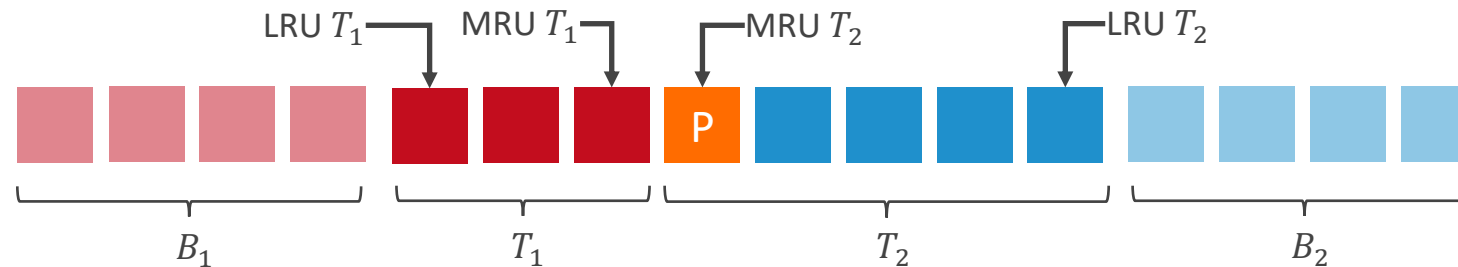
- *Adaptation parameter* $p \in [0, c]$
 - Attempt to keep p pages in T_1 and $c - p$ pages in T_2 .
 - Adapt value of depending on the workload.
 - Initially $p = 0$ and T_1, T_2, B_1, B_2 are empty.

Cache hit

- Case 1: **Page P** is either in T_1 or T_2 .

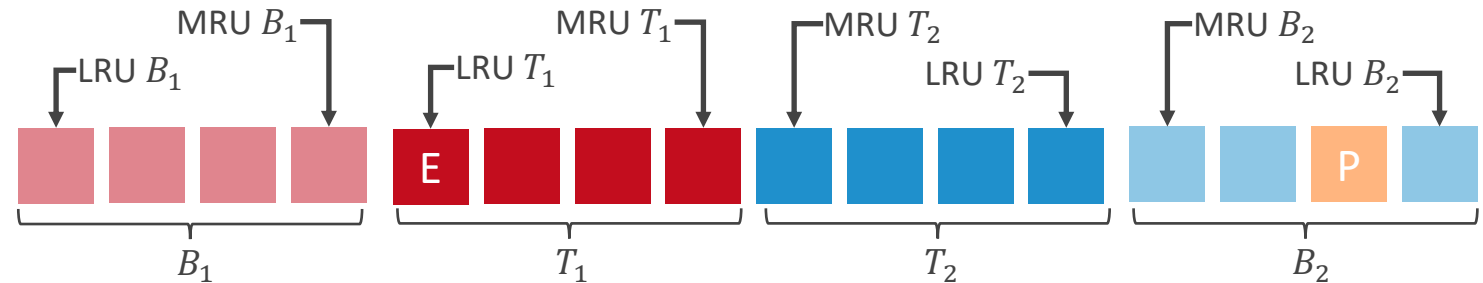


- Move **page P** to MRU of T_2 (page is now frequent).

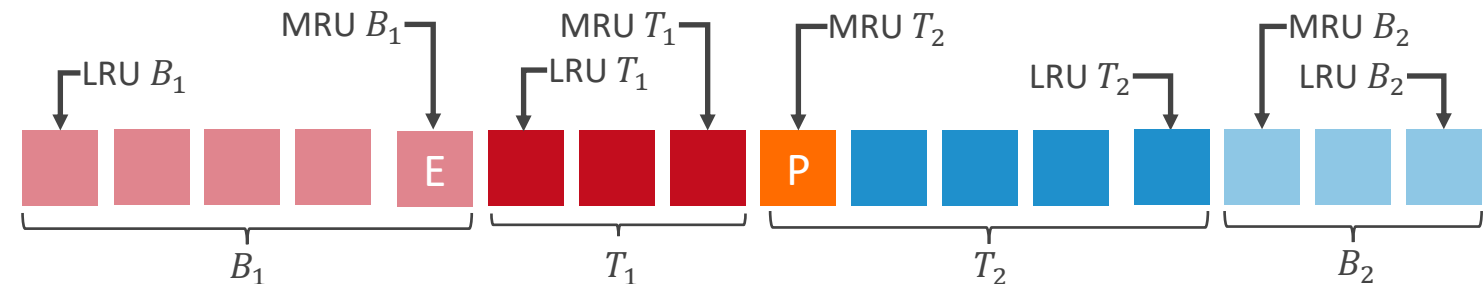


Cache miss but tracked in bottom lists

- Case 2: **Page P** is either in B_1 or B_2 .

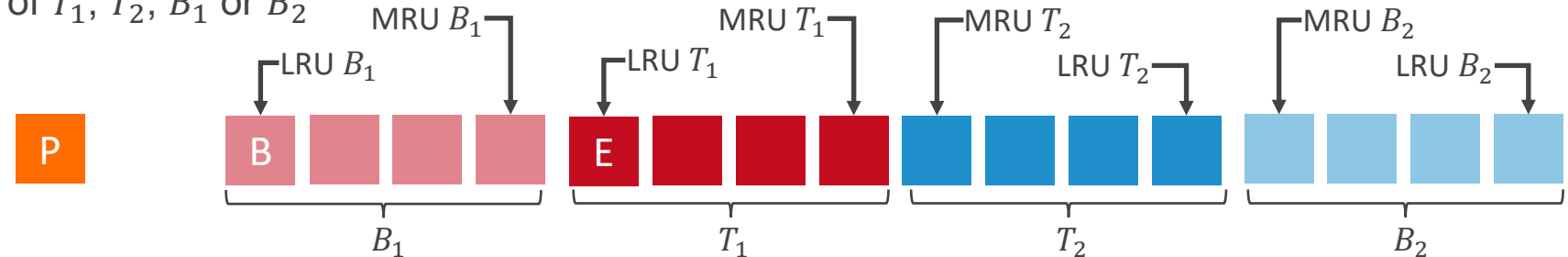


- Evict a **page E** from T_1 or T_2 (next slides).
- Adapt parameter p (next slides).
- Load data of **page P** into the cache.
- Move **page P** to MRU of T_2 (page is now frequent).

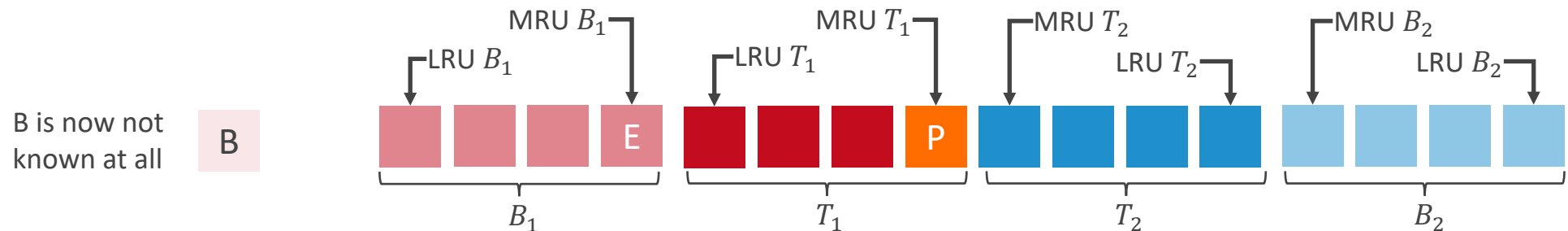


Cache miss and not tracked in bottom lists

- Case 3: **Page P** is not in any of T_1 , T_2 , B_1 or B_2



- Remove a tracked **page B** from B_1 or B_2 (next slides).
- Evict a **page E** from T_1 or T_2 (next slides).
- Load data of **page P** into the cache.
- Move **page P** to MRU of T_1 (page is now recent).



Eviction of pages from the cache

- Evict a **page E** from T_1 or T_2 to make space for a new **page P** (case 2 or case 3).
- Case a:
 - T_1 is not empty AND
 - $|T_1| > p$ OR
 - **Page P** is in B_2 AND $|T_1| = p$
 - Remove LRU of T_1 from the cache and move tracked page to MRU of B_1 .
- Case b:
 - Remove LRU of T_2 from the cache and move tracked page to MRU of B_2 .

Adjustment of size of top lists

- Adjust parameter p when a **page P** was not in the cache but tracked (case 2).

- Case A: **Page P** was in B_1
 - Update $p = \min\{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ \frac{|B_2|}{|B_1|} & \text{otherwise} \end{cases}$

- Case B: **Page P** was in B_2
 - Update $p = \max\{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ \frac{|B_1|}{|B_2|} & \text{otherwise} \end{cases}$

Removal of tracked pages from the bottom lists

- When **page P** was not known in the cache at all (case 3).
 - If bottom lists are full we have to make room to evict a page.
- Case I: $|T_1 \cup B_1| = c$ and $|T_1| < c$
 - Remove LRU from B_1 .
- Case II: $|T_1| = c$ and $|B_1|$ is empty
 - Evict LRU from T_1 from the cache.
- Case III: $|T_1 \cup B_1| < c$ and $|T_1 \cup T_2 \cup B_1 \cup B_2| = 2c$
 - Remove LRU from B_2 .

Full algorithm

Figure 4 in Megiddo and Modha: *ARC: A self-tuning, low overhead replacement cache*, USENIX 2003,
<https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>

ARC(c)

INPUT: The request stream $x_1, x_2, \dots, x_t, \dots$.

INITIALIZATION: Set $p = 0$ and set the LRU lists T_1 , B_1 , T_2 , and B_2 to empty.

For every $t \geq 1$ and any x_t , one and only one of the following four cases must occur.

Case I: x_t is in T_1 or T_2 . A cache hit has occurred in ARC(c) and DBL($2c$).

Move x_t to MRU position in T_2 .

Case II: x_t is in B_1 . A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \min \{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ |B_2|/|B_1| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_1 to the MRU position in T_2 (also fetch x_t to the cache).

Case III: x_t is in B_2 . A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \max \{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ |B_1|/|B_2| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_2 to the MRU position in T_2 (also fetch x_t to the cache).

Case IV: x_t is not in $T_1 \cup B_1 \cup T_2 \cup B_2$. A cache miss has occurred in ARC(c) and DBL($2c$).

Case A: $L_1 = T_1 \cup B_1$ has exactly c pages.

If ($|T_1| < c$)

Delete LRU page in B_1 . REPLACE(x_t, p).

else

Here B_1 is empty. Delete LRU page in T_1 (also remove it from the cache).

endif

Case B: $L_1 = T_1 \cup B_1$ has less than c pages.

If ($|T_1| + |T_2| + |B_1| + |B_2| \geq c$)

Delete LRU page in B_2 , if ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$).

REPLACE(x_t, p).

endif

Finally, fetch x_t to the cache and move it to MRU position in T_1 .

Subroutine REPLACE(x_t, p)

If (($|T_1|$ is not empty) and (($|T_1|$ exceeds the target p) or (x_t is in B_2 and $|T_1| = p$)))

Delete the LRU page in T_1 (also remove it from the cache), and move it to MRU position in B_1 .

else

Delete the LRU page in T_2 (also remove it from the cache), and move it to MRU position in B_2 .

endif

Additional notes

- Once the cache is full, size of $T_1 \cup T_2$ does not change $\rightarrow$ Same amount of data in the cache.
- B_1 and B_2 only track page meta data (resource ID, page number) but not binary data.
- If a requested **page P** is known *anywhere* in the lists, it goes to MRU of T_2 (it is now frequent).
- Otherwise it goes to MRU of T_1 (it is now recent).
- Pinning and prefetching complicate the algorithm a little bit.

Pinning in the ARC

- When a page is to be evicted from T_1 or T_2 , the least recently used *non-pinned* page is evicted.



- If one list (for example T_1) is completely pinned, then the least recent non-pinned element from the other list is evicted.
 - p is not adjusted in this case!
- If both pages are completely pinned, throw an exception.

Prefetching and the ARC

- We want to load pages (add them to the cache) without considering the first retrieval a cache hit.
 - Useful when scanning a table.
 - But retrieving a prefetched page during the table scan should not make the page frequent.
- Adaptation: New page is moved to MRU of T_1 (recent list). First retrieval does not move it to MRU of T_2 (frequent list).
 - Only subsequent retrievals do.

PageCache API

PageCache API

- Get a page from the cache
- Add a page to the cache
- Unpin pages
- House keeping

Get a page from the cache

- `CacheableData getPage(int resourceId, int pageNumber)`
 - Returns the page if it exists or null if it doesn't.
 - Pages that have been retrieved once before, move to MRU of T_2 .
- `CacheableData getPageAndPin(int resourceId, int pageNumber)`
 - Increase pinning counter of retrieved page.
 - Pages cannot be evicted unless pinning counter is 0.

Add a page to the cache

- `EvictedCacheEntry addPage(CacheableData newPage, int resourceId)`
 - Adds a page to the cache (MRU of T_1 or T_2).
 - Evicts a page to make space, adjusts parameter p if necessary.
 - Returns the evicted page (including the wrapped byte array) or an `EvictedCacheEntry` object without an evicted page *but* with a wrapped byte array.
 - This is switched with an I/O buffer in the `BufferPoolManager`.
- `EvictedCacheEntry addPageAndPin(CacheableData newPage, int resourceId)`
 - Increase pinning counter of added page.
 - Subsequent retrievals will move the page to MRU of T_2 (no prefetching logic).

Unpin pages

- `void unpinPage(int resourceId, int pageNumber)`
 - Reduce pinning counter for a page.
- `void unpinAllPages()`
 - Set all pinning counters to 0.

House keeping

- `int getCapacity()`
- `CacheableData[] getAllPagesForResource(int resourceId)`
 - Retrieve all pages for a table or an index.
 - Counts as a normal retrieval, i.e., pages are moved to MRU of T_2 unless they have been prefetched.
- `void expelAllPagesForResource(int resourceId)`
 - Remove all pages for a table or an index.
 - Ignores pinning.
 - Byte arrays must remain in the cache.

Exercise 2

Exercise 2

- Implement the PageCache interface.
- Implement the factory method createPageCache.

Tests and points

Test name	Points	Description
testAddPage	1	Check that adding an existing page throws an exception.
testGetPage	2 + 1	Check that cached pages are returned with the correct byte array (2 points) and that null is returned for uncached pages (1 point).
testPinning	2 + 2	Check that pages are not evicted when they are pinned (2 points) but can be evicted when they are unpinned (2 points).
testGetCapacity	0	Check that we track the correct capacity.
testCacheBehavior	2 + 2	Check correct handling of scan resistance (2 point) and handling of top and bottom lists (2 points).
testRetrievingExpellingAll	1 + 1	Test retrieving (1 point) and expelling (1 point) pages from a certain resource.

Course registration

- Register by Monday morning (November 6, 9:00) in MOSES
- **We cannot help you if you miss this deadline!**